2013 Third Professional Year Project

A remote lab implementation using SAHARA

Nick Wareing

Supervisors: Steve Weddell and Phil Bones

Electrical & Computer Engineering
University of Canterbury
New Zealand
September 2013

A remote lab implementation using SAHARA

Abstract

A remote laboratory system has been designed to allow students to test microcontroller code on model helicopters and view its performance without physically being in a laboratory. They access a web based system, running open-source SAHARA software and are able to remotely upload a compiled program onto a Stellaris LM3S1968 microcontroller. Students view the helicopter on a live video feed and have virtual button presses sent to pre-defined pins of the microcontroller for their program to respond to. Remote resetting of the microcontroller is also supported. An embedded helicopter control program has also been written, for demonstration purposes. Two helicopters are currently accessible remotely and the software is designed to scale easily. The system has proven robust over hours of testing. It costs $1210.10 to setup each helicopter laboratory.

1 INTRODUCTION

This report describes the design and implementation of a remote laboratory system for the University of Canterbury. The laboratory allows Electrical Engineering students to upload compiled C code and use it to control an electric helicopter. They are able to gain visual feedback on the performance of their code via a live video stream.

The central premise of remote labs is that universities around the world can install and host physical experiments which can be accessed by students over the Internet [1][2][3][4]. The idea holds great promise because it can enhance collaboration between universities (by the sharing of expensive lab resources)[5], and moreover, allow students to work from wherever they are, at whatever time of day suits. Remote labs can thus help flatten the demand for expensive lab resources and software is able to impose limits on the safe use of equipment.

As public access is not required to a remote laboratory, they are more secure, reducing problems associated with theft, vandalism and damage to equipment through inappropriate use. Student safety is also improved through physical isolation from hazardous equipment. These factors may have economic benefits in the form of reduced occupational health and safety (OSH) compliance costs and reduced insurance premiums.

Until recently, the technological challenges involved in real-time Internet communication have limited the growth of remote laboratories. However, recent improvements and innovations in web technologies have rapidly begun to remove these impediments. The University of Technology, Sydney (UTS)[1] has already implemented a fully featured remote laboratory system.

In 2012 the Electrical and Computer Engineering department at the University of Canterbury offered a course, ENCE361[2], entitled Embedded Systems. One of the projects, 'Fun with Avionics', required students to write an embedded program to control the flight of a small model helicopter. The helicopter was fixed on a custom built stand which allowed it to rise, fall, and turn through a range of motion (see Fig. 1). Students purchased a Stellaris development board with a Cortex-M3 microcontroller, which included a development platform for compilation, programming, and debugging. The helicopter stand output a voltage proportional to the height of the helicopter. Students were required to control the rotor using pulse width modulation (PWM) signals.

This assignment has proved to be both challenging and interesting for students. It provides a platform to apply embedded programming concepts such as interrupts, implementing proportional, integral, derivative (PID) control, and creating a small real time operation system (RTOS). Due to a limited number of helicopters, and associated challenges, however all students were not able to gain equal access to the laboratory equipment. The remainder of this report discusses how the 'Fun with Avionics' assignment has been converted to a remote laboratory format to help solve some of these issues.

The SAHARA Labs software framework[3], developed at UTS was chosen for the project. Our configuration of SAHARA is described in the following section.

[1] http://www.uts.edu.au/

[2] http://www.canterbury.ac.nz/courses/

[3] http://sourceforge.net/projects/labshare-sahara/

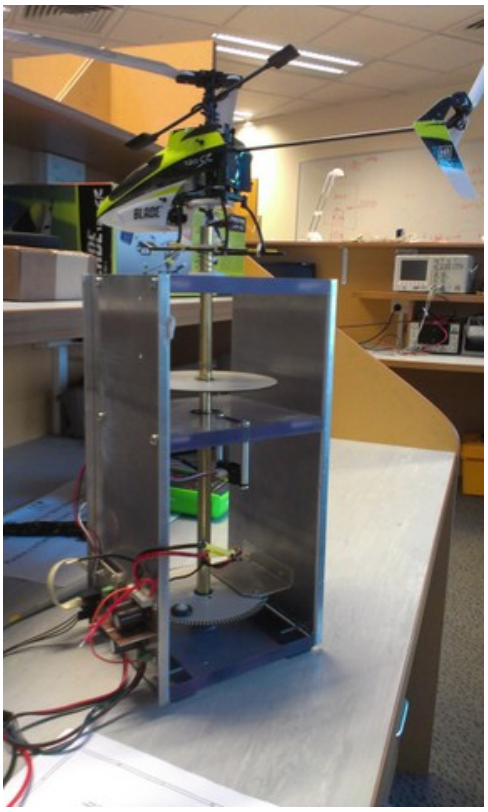

Figure 1: The helicopter is fixed in a stand which allows a range of motion. The stand has an optical sensor to detect the height of the helicopter.

The various design decisions required for the lower level components of the rig are explained next. We then present our example helicopter control program. Finally, the current state of the system is described in detail and future plans are presented. The appendices contain a guide to aid technicians in adding additional rig clients.

Before we delve into the technical detail we will define a number of terms, specific to remote laboratories that will be used throughout the paper:

SAHARA An open-source framework for developing generic remote laboratories.

Rig-Client The SAHARA software application that provides an abstraction to rigs.

Heli-Rig The customised version of the Rig-Client, providing functions to communicate with the helicopter, microcontroller and associated hardware.

Rig A single instance of a laboratory, referring to both the hardware and software. In this case, one rig refers to one helicopter.

For this project to deliver the required experience to students and be successful, the following must be achieved:

1. Two functioning helicopter rigs.
2. The ability to respond to 'virtual' button presses.
3. The ability to upload programs onto the microcontroller remotely.
4. The ability to view the helicopter on a video stream.

5. The system must maintain robust behaviour at all times.

Throughout the remainder of this report, *italics* will be used whenever we refer to elements of code we have written such as class, method, function or variable names.

2 SAHARA

2.1 SAHARA Components

A number of software tools have been created to assist with the development of remote laboratories. UTS' SAHARA, MIT's iLab[6] and Lab2Go[7] are some prominent examples.

The decision to use SAHARA was made in large part for non-technical reasons - Dr Steve Weddell has established a working relationship with the remote labs team at UTS and so it made sense to use their system. Additionally, it has all of the features required to support our desired remote helicopter laboratory.

SAHARA is a generic platform for designing customised remote labs. It is developed and maintained by a team at UTS. SAHARA consists of three separate, but interconnected components:

1. Web Interface
2. Scheduling Server
3. Rig Client

The interactions between these components in the context of the our completed remote laboratory is detailed in the top section of Fig. 2.

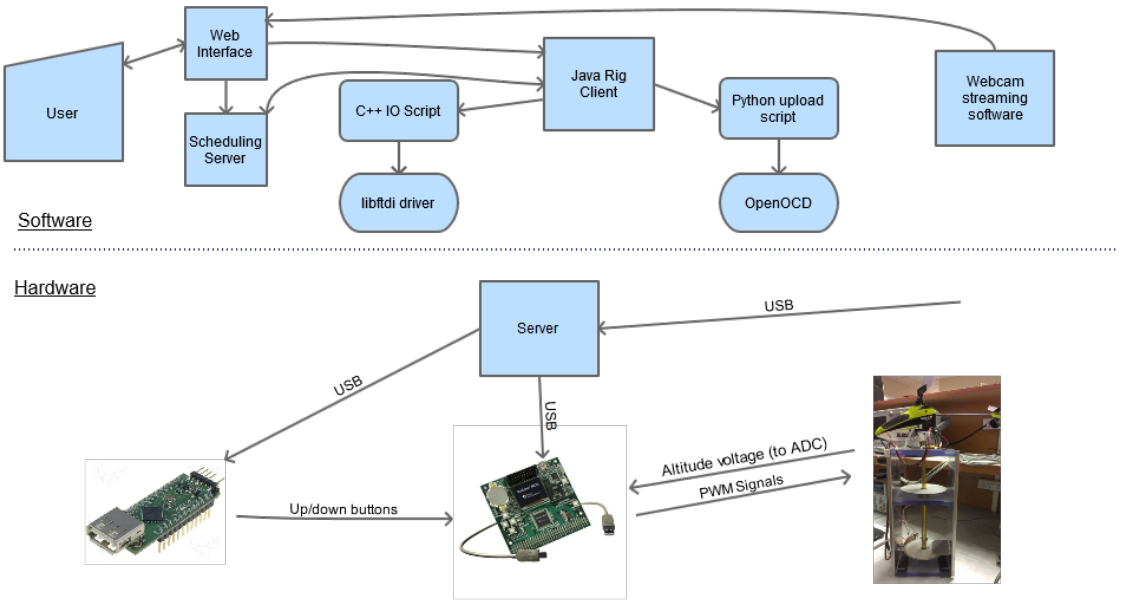

Figure 2: Interactions between various software and hardware components in the remote micro-controlled helicopter laboratory.

2.1.1 Web Interface

The web interface is the platform presented to students to interact with the physical lab. It is coded in PHP (using the Zend Framework[4]), along with HTML, CSS and JavaScript. It allows students to authenticate with the system using either a MySQL database or the university's LDAP servers. Queueing and reservation functions are also supported. The SAHARA library contains a number of 'elements' which can be placed on a customised HTML5 page to support newly designed rigs.

2.1.2 Scheduling Server

The scheduling server forms the core of SAHARA. It is written in Java and needs to connect to a MySQL database. The scheduling server provides a web interface which lab technicians and lecturers can use to allocate access to specific rigs or entire classes of rigs.

2.1.3 Rig Client

The Rig Client is the component that receives user input and communicates with hardware. It is usually installed on a machine that is physically close to the laboratory equipment to reduce the amount of cabling required. Three Rig Client control types are provided by SAHARA [8]:

1. Batch Control: The rig operates using batch instructions uploaded by the user. No other intervention is required.

2. Peripheral Control: A rig has a controller that is 'outside' of SAHARA

3. Primitive Control: The rig is controlled directly by the Rig Client.

Of these options, batch control is obviously not suitable because further interaction is required after program upload to control and monitor the helicopter.

A remote desktop connection is an example of peripheral control. In this case the Rig Client has no direct control over the hardware and is only responsible for assigning users, running tests and detecting rig activity. The helicopter laboratory could be developed in this way by simply installing the Code Composer Integrated Development Environment (IDE) on the remote server and allowing students to develop their code on the server. A desktop program could be written to change the state of the various GPIO lines of the microcontroller via a USB device.

The peripheral control approach is problematic however. Writing code is a time intensive process, so having this done on the remote laboratory side would compromise the system's ability to share access around. Moreover, students would likely be unreceptive to being logged out when in the middle of writing a complex function. Security is also compromised by giving students remote desktop access.

The final option is primitive control. This is typically implemented within the web interface. Actions are routed through to methods of the customised Rig Client class as shown in Fig. 3. Primitive control also provides users with a more elegant and streamlined experience. Security is enhanced because access is only through the web browser. The compromise is that additional development time is required. On the balance of these reasons, primitive control was chosen.

[4]http://framework.zend.com/

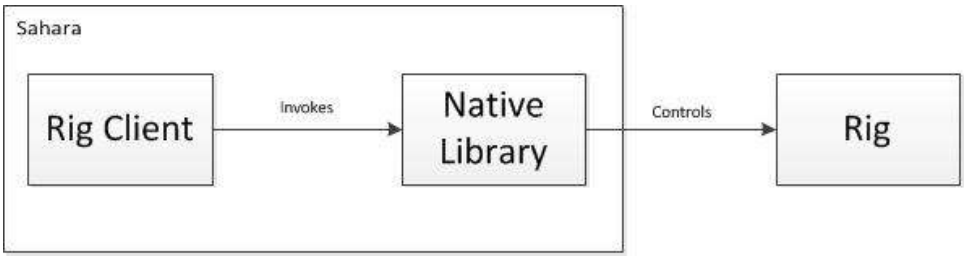

Figure 3: Block level diagram of primitive control [8].

2.2 SAHARA Installation

The following steps were taken to install the three components of SAHARA. Firstly, a standard desktop machine (4GB RAM) was provisioned with the Ubuntu[5] operating system.

In order to install SAHARA the following prerequisites were installed:

- Java 6 (openjdk-6-jdk)
- Alien (to allow the installation of SAHARA RPM packages, designed for Red Hat Linux, on Ubuntu).
- LAMP (a commonly installed stack of web sever technologies, the acronym expands to Linux, Apache, MySQL and PHP). Ubuntu makes this install straightforward using the *tasksel* and *lamp-server* packages.

The Rig Client, Rig Client Commons (a further set of optional abstractions) and Scheduling Server packages were then installed using Alien. Finally, the Web Interface files were copied to the root of the Apache document folder.

2.2.1 Deployment

We quickly determined that a quick code-to-test cycle was going to be crucial to both the project's success and to providing an enjoyable and efficient development experience. The two SAHARA components that would require significant customisation are the Web Interface and Rig Client. The procedure for deploying changes to each component are different. Changes to the web interface simply require the affected files to be copied to the correct directory on the server. Users then see the new changes when the page is next loaded or refreshed.

Each change to the Rig Client requires the Java class to be recompiled using an Ant build script (similar to make). In order to deploy this class, the Rig Client service must be stopped and then started again when the new class is copied across. If done manually this deployment is a tedious process and takes time away from doing 'real' development work. To make better use of my time, I wrote a deployment script in Python, using the Fabric library. Fabric[6] is designed to help streamline deployment and system administration tasks over SSH.

In the later stages of the project, when extra servers were provisioned to host extra Rig Clients (so that multiple helicopters could be in use), it was a simple matter of adding their host names to this script.

[5]http://www.ubuntu.com/
[6]http://docs.fabfile.org/en/1.7/

Git[7] has been used for version control throughout the project. The repository is hosted on the department's GitLab sever at `https://eng-git.canterbury.ac.nz/david.vanleeuwen/labshare`.

3 RIG CUSTOMISATION

3.1 Rig Client Class

The Rig Client is coded in Java, and utilises several third party libraries. In order to design a customised rig a developer creates a new Java class which extends the *AbstractControlledRig* class. The Rig Client program loads in this class at runtime (after specifying its presence in a configuration file). Our customised *HeliRig* class has four main features:

1. Facility to take the compiled student program from its temporary upload directory (or the demo program) and flash it onto the Stellaris Microcontroller.
2. Facility to channel virtual button presses to specific pins of a USB to parallel device.
3. Reset functions which the Rig Client runs after a student finishes using the rig, to ensure it is ready for the next user.
4. Test functions which run periodically by the Rig Client to determine if the rig remains in a working order.

These features were developed in an iterative manner, with each significant change being tested and then committed to Git.

3.2 Web Interface Customisation

SAHARA provides a number of 'elements' which can be placed within an HTML page to develop a customised web interface for a specific rig type. This meant it was a relatively simple exercise to build up the page. The majority of the development effort was in *heli.js* which responds to user actions on the web page and invokes methods of the *HeliController* class. The actions are sent to the appropriate server asynchronously in JSON (JavaScript Object Notation) format via an HTTP GET. This means that there are no page reloads - the web site behaves as though it were a desktop application.

The completed web interface, as show in Fig. 4 includes a connection to the video stream along with buttons to perform upload of binaries and to control the microcontroller GPIO lines.

3.3 Virtual Buttons

The next step was to investigate how GPIO signals would be sent to the Stellaris board. An FTDI FT245R USB to Parallel-FIFO chip[8] was chosen for this purpose.

[7] http://git-scm.com/
[8] http://www.ftdichip.com/Products/ICs/FT245R.htm

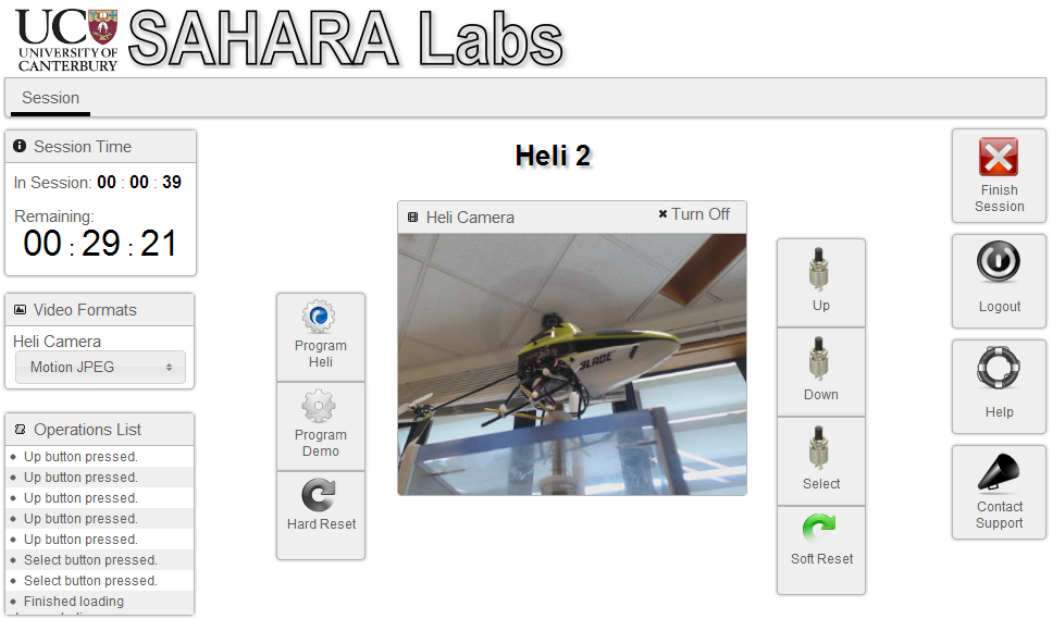

Figure 4: The web interface in action. The buttons of the left of the video stream allow program binaries to be uploaded and allow users to reset the microcontroller via JTAG. The buttons immediately right of the video send signals to allocated GPIO lines of the microcontroller.

A number of other options were considered, including an Arduino board[9] and a purpose built USB to GPIO module. However the FTDI has the best combination of features and price. It meets all our requirements with the following features [9]:

1. Complete USB device mode protocol handled on-chip without custom programming.
2. Complete USB hardware on-chip, including USB resistors.
3. 8 GPIOs available - this will allow for future expansion.
4. USB suspend and resume support, to switch device to low-power mode when not in use.
5. Integrated level converter and 5 V, 3.3 V, 2.8 V and 1.8 V totem-pole output, so it can interface with most standard microcontrollers. In this case, 5V logic is used to interface with the Stellaris.
6. No additional crystal or oscillator required.

In order to communicate with the FTDI chip, the open source libftdi[10] library is used. Initially, a simple breadboard circuit was constructed where an LED was connected to one of the GPIO pins of the FTDI chip, as shown in Fig. 5. A simple program was written and run to test that:

1. The toolchain and libftdi library worked.
2. The circuit was feasible.

[9] http://www.arduino.cc/
[10] http://www.intra2net.com/en/developer/libftdi/repository.php

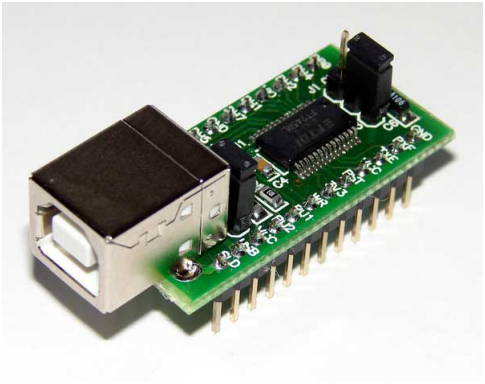

Figure 5: The GPIO signals are sent to the Stellaris microcontroller using an FTDI device.

3.3.1 Reset Functionality

The 'soft reset' button, triggered by one of the virtual inputs discussed above relies on the user's program to interrupt on this signal and perform a software reset.

The 'hard reset' button, on the other hand, causes the OpenOCD daemon to send a reset command via JTAG, and so will work regardless of the program loaded on the Stellaris device.

The Python script in Listing 1 shows how the 'hard reset' functionality is implemented. The script is called from the Java program as shown in Fig. 6. It can be easily adapted in the future to send any other OpenOCD arguments.

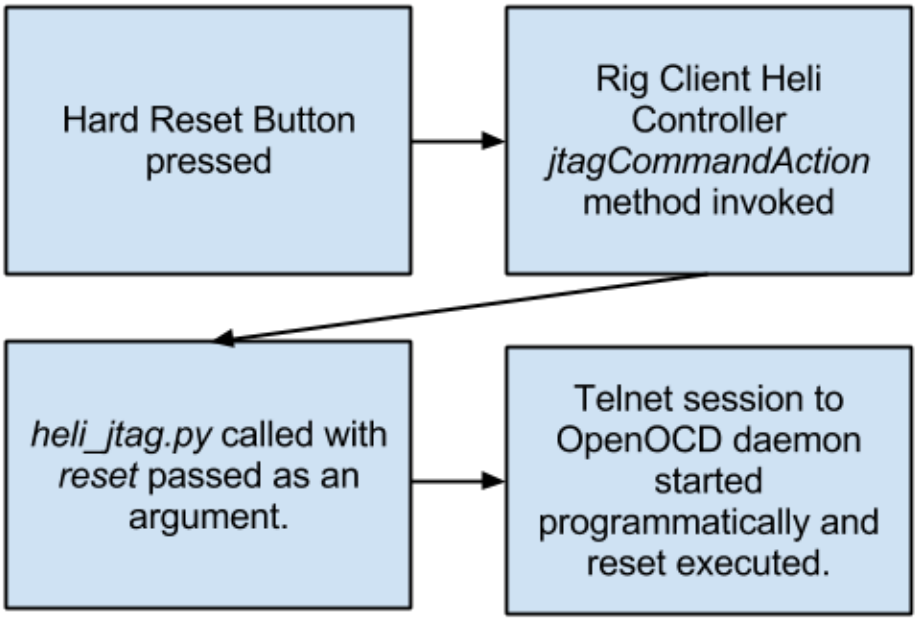
Hard Reset Button pressed
Rig Client Heli Controller *jtagCommandAction* method invoked
heli_jtag.py called with *reset* passed as an argument.
Telnet session to OpenOCD daemon started programmatically and reset executed.

Figure 6: Block diagram of a 'hard reset' execution. The button calls the *jtagCommandAction* method in the customised Rig Client. This method then invokes a Python script which executes the command inside an OpenOCD telnet session.

Listing 1: Python script to send OpenOCD commands

```
import pexpect
import argparse
import os
import sys

def main(**kwargs):
    jtag_command(kwargs['command'])

def jtag_command(command):
    child = pexpect.spawn('telnet localhost 4444')

    child.sendline(command)
    child.expect('>')

    child.sendline('exit')

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Execute an OpenOCD command')
    parser.add_argument('command', type=str, choices=['halt','resume','reset'], \\
        help='Provide an OpenOCD command')
    args = parser.parse_args()
    main(**vars(args))
    sys.exit()
```

3.4 Remote Programming

To program the Stellaris, we have chosen to use OpenOCD[11], one of the most popular open-source in-system programmers. Once installed, it is run from the command line with the location of the configuration file given as an argument. It then verifies the JTAG scan chain specified and if successful, starts as a daemon, waiting for connections from telnet clients or the GDB debugger. The most recent versions come with configuration scripts for the Stellaris EKS-LM3S1968, so it is trivial to get working.

The UTS team opted to use a shell script for their bitstream upload script, however we have chosen to use a simple Python program. This is mainly due to our preference for Python, along with the fact that the language has evolved such that there are libraries to accomplish most system level tasks. In this case we are using the pexpect module (short for Python Expect)[12] to communicate with OpenOCD via a telnet session. Expect is an extension to the Tcl scripting language, allowing users to easily create scripts which talk to interactive applications. Here, it is used to start a telnet session to OpenOCD, reset the Stellaris microcontroller and flash the '.bin' file to the microcontroller. Additionally, the Python argparse library is used to process the command line arguments sent from the Java Rig Client instance. The script has been successfully verified and tested in isolation.

[11] http://openocd.sourceforge.net/
[12] http://www.noah.org/python/pexpect/

3.5 Prototype Stellaris Program

We now turn our discussion to the embedded C program students must write to control the helicopter. Recall that button presses on the web interface result in an asynchronous function call to the Heli Rig class. The Java function checks the argument passed to it is a valid integer and then calls a C function via the Java Native Interface (JNI)[13]. The JNI allows Java code running in the Java Virutal Machine (JVM) to call, and be called by native applications written in other languages.

We use the JNI here because the FTDI board is controlled with a C library. The C code we have written performs a number of checks to see that the FTDI board exists and is connected to the server (if it is not, the rig is taken offline). It then pulls a corresponding pin low for a predefined period of 200ms.

The upshot of this design from a user's perspective is that their program must be configured to interrupt on the falling edge of the attached GPIO pins. This information is provided in a user's manual. This document also provides a short code extract to perform this GPIO interrupt initialisation. With this done, students can work on the rest of their program with no thought to its being used in a remote lab context.

The virtual buttons are mapped to GPIO inputs of the Stellaris development board as follows (all pins are on Port B):

Button	Pin Description	Pad #
Select	PB4/C0-	1
Up	PB5/C1-	3
Down	PB6/C0+	4
Reset	PB1/CCP2	18

The GPIO peripheral is initialised to interrupt on the falling edge of these pins with the code shown in Listing 2:

Listing 2: Initialising the required GPIO pins

```
1  void
2  initialisePortB (void) {
3      // GPIO PortB must be enabled for configuration and use.
4      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
5
6      // Set pins 5 & 6 as input, SW controlled.
7      GPIOPinTypeGPIOInput(GPIO_PORTB_BASE, GPIO_PIN_1 | GPIO_PIN_4 |
8          GPIO_PIN_5 | GPIO_PIN_6);
9
10     // Register the port-level interrupt handler.
11     GPIOPortIntRegister(GPIO_PORTB_BASE, portBIntHandler);
12
13     // Make pins 5 & 6 a "both edges" triggered interrupt.
14     GPIOIntTypeSet(GPIO_PORTB_BASE, (GPIO_PIN_1 | GPIO_PIN_4 |
15         GPIO_PIN_5 | GPIO_PIN_6), GPIO_FALLING_EDGE);
16
17     // Set the interrupt priority to '0x40'.
```

[13] http://docs.oracle.com/javase/6/docs/technotes/guides/jni/

```
18      IntPrioritySet(INT_GPIOB, 0x40);
19
20      // Enable the pin interrupts.
21      GPIOPinIntEnable(GPIO_PORTB_BASE, (GPIO_PIN_1 | GPIO_PIN_4 |
22          GPIO_PIN_5 | GPIO_PIN_6));
23
24      IntEnable (INT_GPIOB); // Note: INT_GPIOB defined in inc/hw_ints.h
25  }
```

Students are also told that in order to control the helicopter, they must configure the ADC and PWM peripherals as follows:

- The helicopter's altitude reading is connected to **ADC0** of the Stellaris board.
- The helicopter's main rotor is connected to **PWM1** of the Stellaris board.

To give students confidence in the remote lab system and to allow us to perform tests during its development, a demonstration helicopter control program has been written. Students do not have access to the source code of this program, but are able to press the 'Program Demo' button on the web interface at any time to have the binary loaded onto the Stellaris board.

The demonstration program was designed to meet all of the specifications required in the student's assignment. A real-time operating system, FreeRTOS[14] was used to break the code into a number of tasks as shown in Fig. 7. Communication between tasks is by way of queues and semaphores.

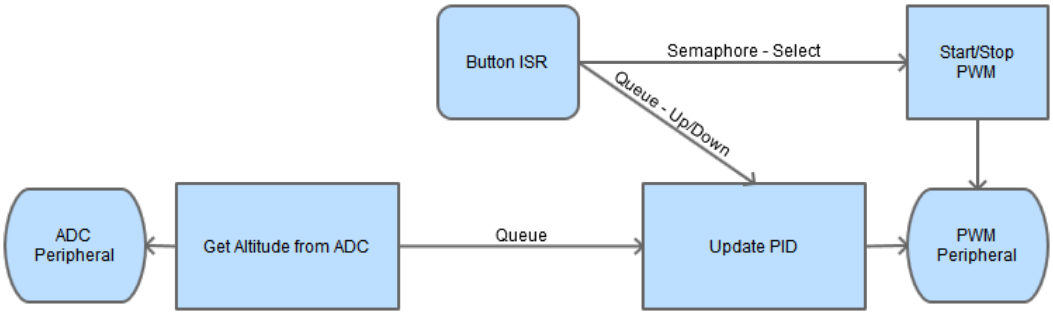

Figure 7: The helicopter control program was broken down into a number of FreeRTOS tasks.

3.6 Video Feedback

One important point to note is that the video streaming and Rig Client have no connection with each other. One must simply ensure that the correct video formats are being continuously streamed, using any suitable software.

We chose to use the Logitech C920 Webcam for video capture, and ffmpeg[15] as our streaming and transcoding solution. ffmpeg streaming can be initiated from the command line. For example, the following invocation (Listing 3) starts streaming in the formats described by *ffserver.conf*:

[14] http://www.freertos.org/
[15] http://www.ffmpeg.org/

Listing 3: Initialising the virtual button pins

```
ffserver -f ffserver.conf & ffmpeg -v quiet -r 5 -s 320x240 -f video4linux2 -i\
/dev/video0 http://localhost:7070/feed1.ffm
```

4 SYSTEM SUMMARY AND BUDGET

4.1 Summary of Software

The following software is used in the system. In order to reduce cost, only open source software was chosen.

Software	**Use**	**License**
SAHARA	Remote Labs Framework	BSD
Ubuntu	Server Operating System	GNU GPL
Java JDK 6	Java Virtual Machine and Compiler	GNU GPL
Apache	Web Server	Apache License
MySQL	Scheduling Server Database	GNU GPL
PHP	PHP Interpreter	PHP License
libftdi	FTDI USB driver with bitbang mode	LGPL
OpenOCD	In-system programming for Stellaris device	GNU GPL
ffmpeg	Video transcoding and streaming of helicopter	GNU GPL
Python	Interaction with OpenOCD telnet daemon	Python License
FreeRTOS	Real time operating system for demo	GNU GPL

4.2 Summary of Hardware (per rig)

The hardware costings are listed below. It can be seen that the cost scales linearly with the number of rigs provisioned.

Item	**Supplier**	**Price (NZD)**
Model Helicopters	Acorn Models	$250.00
Logitech C920 Webcam	Ascent	$124.32
Stellaris EKS-LM3S19	Element 14	$103.83
FTDI UM245R Board	Element 14	$31.95
10A Bench Power Supply	UC ECE	$200
2x Headless Computers	UC ECE	$500
Total:		$1210.10
Total (for 2 rigs):		$2220.20

5 DISCUSSION

5.1 Performance

The system as it stands meets or exceeds all of the requirements specified. Testing has shown the system to be robust with the helicopters having been flown remotely for many hours. Although not the immediate focus of this project, a number of

mechanical changes have been made to the helicopter stands to further enhance reliability:

1. Springs were added to prevent the helicopter being damaged when the motors are stopped mid-flight. These have performed well and greatly enhance robustness.

2. The power cables for the main rotor are now routed down the shaft of the stand rather than hanging freely from the helicopter. This prevents them from wrapping around the helicopter or being caught in the blades.

Latency in the video feed is one issue that had the potential to cause problems with this project because it is important that students are able to press a button and see the helicopter respond quickly. We have tested the rig (hosted at the University of Canterbury in Christchurch, New Zealand) from Canberra, Australia and have found the delay in the video to be acceptable, at approximately one second.

The software has been developed to achieve graceful failure wherever possible. In most cases, if an error is encountered, such as the USB-to-parallel chip being disconnected, an informative error message will be written to a log file, the user logged out of the system, and the rig automatically placed in an offline state until a technician resolves the problem. Staff are notified of problems automatically via email.

The system is highly scalable. To add a new rig (helicopter) to the remote laboratory a new computer is provisioned and the SAHARA software installed. With a webcam, USB to parallel chip and Stellaris microcontroller connected, all that remains is for the IP address of the scheduling server to be specified in the new rig's configuration file. Registration then occurs automatically. A guide to aid technical staff in this process has been written and can be found in Appendix A.

5.2 Security

Allowing students to upload executable programs to university servers is an inherently security sensitive operation. The target audience of this facility are predominantly young electrical engineering and computer science students, the type of individuals who like to push the bounds of whatever system they are using! From a network and computer security standpoint this means that the software should be carefully designed to be resistant to exploits and that the user's actions should be tightly controlled. Our design puts security at the forefront by choosing a SAHARA primitive controller. This means that students access the remote laboratory only through a web interface and it does not expose a Remote Desktop Connection as a number of existing systems have done. Only port 80 is exposed which greatly reduces the vulnerability of the server as there is no way for the end user to open a command shell.

Another major concern is that of running student's code on the microcontroller without performing any sort of verification on it (other than checking that the file extension is that of a valid binary file). For the most part this should not be a problem, as causing a chip to stop working would be a detriment to the student and his classmates. However, if the system were to be expanded to a wider user base, a new system might be put in place where it is the source code that is sent to the

rig server which then runs some sanity checks on its contents before compiling it for the microcontroller.

5.3 Future Plans

5.3.1 Debugging Features

At the present point in time, the only feedback to the student in regards to the performance of their program is the video stream. This has a number of issues [10]. Firstly, it does not give students any debugging information in terms of program execution. Secondly, and perhaps of greater concern is that the onus is on the student to slow the speed of the helicopter or reset the microcontoller if their program is poorly functioning. For example, due to a poor PID control implementation, the helicopter may run at 100% of its altitude range for an extended period of time. This risks burning out the motor. For this reason, a number of extra features should be added in future design iterations.

One such feature would involve the Rig Client interpreting the analogue voltage coming from the helicopter stand which is linearly proportional to the helicopter's height. If this is read in, perhaps through an Arduino device with an analog to digital converter (ADC), the Rig Client could send a signal to a relay circuit to have the power to the helicopter cut, and notify the student that their program needs improvement [11]! This could also be implemented with a programmable logic controller (PLC). A second feature being considered is to use the second serial port of the Stellaris device to send debug information back to the web interface in a text stream to aid students in the development of their programs.

5.3.2 Server Configuration

At present, the web interface and scheduling server are hosted on an average desktop computer. When the system is fully developed, these services are likely to be moved to virtual machines (VMs) on the department's existing server infrastructure. Using VMs can provide a number of advantages including live migration, reduced electricity usage and better distribution of computational load [12].

At present, authentication is local to the scheduling server using a simple configuration with the username and password stored in a MySQL database. When students start using the system it is desirable that they can use their existing University of Canterbury credentials. SAHARA makes this transition to using LDAP (lightweight directory access protocol) simple; changing a few lines of the configuration file is all that is required.

5.4 ENZCON Paper

A research paper has been written and was accepted to the ENZCON 2013 conference [13]. This was presented at Massey University in Auckland on 5th September 2013.

6 CONCLUSION

We have presented a new application of remote laboratories, aimed at undergraduate Electrical Engineering students. A complete system has been developed, utilising the SAHARA remote laboratory framework. It allows students to upload compiled code and monitor its ability to control the flight of a small model helicopter. Currently two rig clients are in operation (two helicopters). The next stage of the project should add more advanced forms of feedback to help students debug their programs. We are pleased with the success of this remote laboratory implementation and are excited about the experience it will provide to future students.

References

[1] D. Ursutiu, C. Samoila, and M. Dabacan, "Cross platform methods in digital electronics engineering education," in *Remote Engineering and Virtual Instrumentation (REV), 2013 10th International Conference on*, 2013.

[2] C. F. Riman, A. El Hajj, and I. Mougharbel, "A remote lab experiments improved model," *International Journal of Online Engineering (iJOE)*, vol. 7, no. 1, pp. 37–39, 2011.

[3] K. Jona, R. Roque, J. Skolnik, D. Uttal, and D. Rapp, "Are remote labs worth the cost? insights from a study of student perceptions of remote labs," *International Journal of Online Engineering (iJOE)*, vol. 7, no. 2, pp. 48–53, 2011.

[4] B. Aktan, C. Bohus, L. Crowl, and M. Shor, "Distance learning applied to control engineering laboratories," *Education, IEEE Transactions on*, vol. 39, no. 3, pp. 320–326, 1996.

[5] P. Orduña, L. Rodriguez-Gil, D. López-de Ipiña, and J. Garcia-Zubia, "Sharing remote labs: A case study." *International Journal of Online Engineering*, vol. 9, pp. 26 – 27, 2013.

[6] M. I. of Technology, "ilabs: Internet access to real labs," 2009. [Online]. Available: http://icampus.mit.edu/projects/ilabs/

[7] D. Zutin, M. Auer, C. Maier, and M. Niederstatter, "Lab2go: A repository to locate educational online laboratories," in *Education Engineering (EDUCON), 2010 IEEE*, 2010.

[8] "Sahara Development Handbook Version 1.0," 2012.

[9] Future Technology Devices Interntional Ltd., "FT245R USB FIFO IC datasheet version 2.12," 2010. [Online]. Available: http://www.ftdichip.com/Products/ICs/FT245R.htm

[10] A. Cardoso, M. Vieira, and P. Gil, "A remote and virtual lab with experiments for secondary education, engineering and lifelong learning courses," *International Journal of Online Engineering (iJOE)*, vol. 8, no. S2, pp. 49–54, 2012.

[11] D. A. Samuelsen and O. H. Graven, "Assessment of the quality of low cost data acquisition equipment for remote lab setups," in *Remote Engineering and Virtual Instrumentation (REV), 2013 10th International Conference on.* IEEE, 2013.

[12] V. Lasky, "Implementing resilient remote laboratories with server virtualization and live migration," in *Remote Engineering and Virtual Instrumentation (REV), 2013 10th International Conference on.* IEEE, 2013.

[13] N. Wareing, S. Weddell, and P. Bones, "A remote lab implementation using SAHARA," *ENZCON Proceedings*, pp. 58 – 63, 2013.

APPENDIX A: Fun with Avionics Heli Rig Technician Guide

August 2013

1 Prerequisites

- Ubuntu installed.
- Scheduling Server installed on another machine.
- Stellaris board, Logitech webcam and FTDI device all connected via USB.
- SSH, Screen and Vim installed.

2 Acquiring the SAHARA Binaries

Download the RigClient and RigClient-Commons RPMs from `http://sourceforge.net/projects/labshare-sahara/files/Installers/r3.2/`.

3 Installing the Rig Client and Libraries

```
telnet ienabler.canterbury.ac.nz 259
sudo apt-get update
sudo apt-get install openjdk-6-jre openjdk-7-jre alien openocd vim
    libftdi-dev git yasm
sudo alien -i RigClient-3.2-0.x86_64.rpm
sudo alien -i RigClient-Commons-3.2-0.x86_64.rpm
```

3.1 Configuring the Rig Client

The easiest way to do this is to copy the configuration files and *HeliRig* class from an existing rig client. The only thing that needs to change is to increment the number of the *Rig_Name* parameter and specify the address of the scheduling server, both in *rigclient.properties*.

The temporary directories for the batch operation also need to be created as follows.

```
sudo mkdir /var/tmp/rigclient
sudo mkdir /var/cache/rigclient
```

4 Installing ffmpeg

```
sudo git clone git://source.ffmpeg.org/ffmpeg.git
cd ffmpeg/
sudo ./configure
sudo make
sudo make install
usermod -a -G video nmw50
```

Replace *nmw50* with your username.

5 Starting everything up

The following assumes the OpenOCD and ffmpeg config scripts are located in your home directory.

```
sudo openocd
```

Ctrl-z bg

```
ffserver -f /etc/ff.conf & ffmpeg -v quiet -r 5 -s 320x240 -f
    video4linux2 -i /dev/video0 http://localhost/webcam.ffm
```

Ctrl-z bg

6 Location of Config Files

6.1 Web Interface

config.ini Located in */var/WI/application/configs*. Contains the settings for LDAP, database email and institution customisation.

6.2 Scheduling Server

schdeulingserver.properties Located in */usr/lib/schedulingserver/conf*.

6.3 Rig Client

rigclient.properties Located in */usr/lib/rigclient/conf*. Contains parameters to uniquely identify a rig, and specify its type.

rigattributes.properties Located in */usr/lib/rigcleint/conf/conf.d*. Contains settings for the video feed including formats, size, port, and the number of cameras.

batch.properties Located in */usr/lib/rigcleint/conf/conf.d*. Contains settings for the temporary storage of uploaded program binaries.

heli.properties Located in */usr/lib/rigcleint/conf/conf.d*. Specifies the location of the various external scripts and example program binaries.

7 FAQ & Troubleshooting

In almost every case, the most useful debugging resource I found was the log files.